

APPLICATION FOR UNITED STATES LETTERS OF PATENT

FOR

**METHOD FOR COLLECTION OF MEMORY REFERENCE INFORMATION  
AND MEMORY DISAMBIGUATION**

Inventor(s): **Daniel M. Lavery  
David C. Sehr  
Rakesh Ghiya**

Prepared by:

BLAKELY SOKOLOFF TAYLOR & ZAFMAN, LLP  
12400 Wilshire Boulevard, 7th Floor  
Los Angeles, California 90025  
(425) 827-8600

"Express Mail" Label Number EL431685346US

Date of Deposit March 29, 2001

*I hereby certify that this paper or fee is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service under 37 CFR 1.10 on the date indicated above and is addressed to the Assistant Commissioner for Patents, Box Patent Application, Washington, D.C. 20231.*

Melanie Besecker 3/29/01  
Melanie Besecker Date

# **METHOD FOR COLLECTION OF MEMORY REFERENCE INFORMATION** **AND MEMORY DISAMBIGUATION**

## **BACKGROUND OF THE INVENTION**

### **Field of the Invention**

5           The present invention concerns compilers in general, and more specifically concerns a method for collecting memory information and using such information to provide memory disambiguation.

### **Background Information**

10           Memory disambiguation is the process of determining the relationship between the memory locations accessed (or possibly accessed) by a pair of loads, stores, and/or function calls. Compilers perform memory disambiguation to ensure correctness and enhance the effectiveness of optimizations and scheduling. For example, the compiler must determine that a load and store never access the same memory location in order to reorder them during code scheduling. In addition, the compiler must determine that two loads always access the same memory location in order to remove the later  
15           redundant load. If the compiler does not have enough information to disambiguate a pair of memory references, it must be conservative, potentially inhibiting an optimization. For processors that exploit high levels of instruction-level parallelism (ILP), conservative memory disambiguation decisions are a significant performance  
20           bottleneck. Current memory disambiguation methods are either too conservative or are inefficient in compile time or memory usage.

          Modern processors face the ever-increasing gap between processor core speeds and memory speeds. Because of this, the effective cost of a load operation may range from single-digit cycles for cache accesses up to hundreds of cycles for  
25           main memory accesses. The best solution to this problem, as always, is to eliminate as many memory references as possible. Large register files make register promotion of very large numbers of locations practical. To hide the latency of those that remain, the

compiler would still like to have maximum freedom to schedule them. Some modern processors, such as the Intel Itanium™ processor, incorporate data speculation to allow scheduling freedom across some data dependencies that would otherwise sequentialize the schedule. However, the data speculation resources are finite and their use subject to certain constraints. It is therefore still of the foremost importance to prove memory references independent whenever possible. Both of these tasks, register variable promotion and scheduling, rely intimately on the best possible memory disambiguation technology.

Many of a compiler's optimizations that rely on memory disambiguation occur in the compiler backend, and interact with a disambiguator in complicated ways. For instance, to generate efficient code for a machine with a single register-indirect addressing mode requires that addresses be lowered to base and offset early in the compilation. Typically, after the program representation is lowered and optimizations are performed, much of the source-level information is lost and the code is transformed in ways that make it more difficult for the compiler to perform memory disambiguation. For example, after optimization an array reference `a[i]` in a source-level loop becomes a register indirect reference off an induction variable that is initialized outside the loop. It then takes a good deal of searching to find out which array is accessed, let alone which element. Another example is that lowering may make disambiguation much more difficult by obscuring such simple facts as two scalar variables that are not contained in the same structure can never conflict. Therefore the disambiguator needs to retain a certain amount of "high-level" information about storage locations.

Relying solely on high-level information, though, may result in missed opportunities as well. Notably, if the program contained pointer arithmetic such as the following fragment, lowered addressing and constant propagation are needed to prove that `s.b` can be registerized across the store whenever `i` is zero. Because of this interaction between disambiguation and optimizations, an effective disambiguator will

need to incorporate information from a variety of semantic levels of the intermediate language (IL).

5        struct { int a, b; } s;  
      int \*p = &s.a;  
      s.b = 0;  
      \*(p + i) = 1;  
      ... = s.b;

10

11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110  
111  
112  
113  
114  
115  
116  
117  
118  
119  
120  
121  
122  
123  
124  
125  
126  
127  
128  
129  
130  
131  
132  
133  
134  
135  
136  
137  
138  
139  
140  
141  
142  
143  
144  
145  
146  
147  
148  
149  
150  
151  
152  
153  
154  
155  
156  
157  
158  
159  
160  
161  
162  
163  
164  
165  
166  
167  
168  
169  
170  
171  
172  
173  
174  
175  
176  
177  
178  
179  
180  
181  
182  
183  
184  
185  
186  
187  
188  
189  
190  
191  
192  
193  
194  
195  
196  
197  
198  
199  
200  
201  
202  
203  
204  
205  
206  
207  
208  
209  
210  
211  
212  
213  
214  
215  
216  
217  
218  
219  
220  
221  
222  
223  
224  
225  
226  
227  
228  
229  
230  
231  
232  
233  
234  
235  
236  
237  
238  
239  
240  
241  
242  
243  
244  
245  
246  
247  
248  
249  
250  
251  
252  
253  
254  
255  
256  
257  
258  
259  
260  
261  
262  
263  
264  
265  
266  
267  
268  
269  
270  
271  
272  
273  
274  
275  
276  
277  
278  
279  
280  
281  
282  
283  
284  
285  
286  
287  
288  
289  
290  
291  
292  
293  
294  
295  
296  
297  
298  
299  
300  
301  
302  
303  
304  
305  
306  
307  
308  
309  
310  
311  
312  
313  
314  
315  
316  
317  
318  
319  
320  
321  
322  
323  
324  
325  
326  
327  
328  
329  
330  
331  
332  
333  
334  
335  
336  
337  
338  
339  
340  
341  
342  
343  
344  
345  
346  
347  
348  
349  
350  
351  
352  
353  
354  
355  
356  
357  
358  
359  
360  
361  
362  
363  
364  
365  
366  
367  
368  
369  
370  
371  
372  
373  
374  
375  
376  
377  
378  
379  
380  
381  
382  
383  
384  
385  
386  
387  
388  
389  
390  
391  
392  
393  
394  
395  
396  
397  
398  
399  
400  
401  
402  
403  
404  
405  
406  
407  
408  
409  
410  
411  
412  
413  
414  
415  
416  
417  
418  
419  
420  
421  
422  
423  
424  
425  
426  
427  
428  
429  
430  
431  
432  
433  
434  
435  
436  
437  
438  
439  
440  
441  
442  
443  
444  
445  
446  
447  
448  
449  
450  
451  
452  
453  
454  
455  
456  
457  
458  
459  
460  
461  
462  
463  
464  
465  
466  
467  
468  
469  
470  
471  
472  
473  
474  
475  
476  
477  
478  
479  
480  
481  
482  
483  
484  
485  
486  
487  
488  
489  
490  
491  
492  
493  
494  
495  
496  
497  
498  
499  
500  
501  
502  
503  
504  
505  
506  
507  
508  
509  
510  
511  
512  
513  
514  
515  
516  
517  
518  
519  
520  
521  
522  
523  
524  
525  
526  
527  
528  
529  
530  
531  
532  
533  
534  
535  
536  
537  
538  
539  
540  
541  
542  
543  
544  
545  
546  
547  
548  
549  
550  
551  
552  
553  
554  
555  
556  
557  
558  
559  
560  
561  
562  
563  
564  
565  
566  
567  
568  
569  
570  
571  
572  
573  
574  
575  
576  
577  
578  
579  
580  
581  
582  
583  
584  
585  
586  
587  
588  
589  
590  
591  
592  
593  
594  
595  
596  
597  
598  
599  
600  
601  
602  
603  
604  
605  
606  
607  
608  
609  
610  
611  
612  
613  
614  
615  
616  
617  
618  
619  
620  
621  
622  
623  
624  
625  
626  
627  
628  
629  
630  
631  
632  
633  
634  
635  
636  
637  
638  
639  
640  
641  
642  
643  
644  
645  
646  
647  
648  
649  
650  
651  
652  
653  
654  
655  
656  
657  
658  
659  
660  
661  
662  
663  
664  
665  
666  
667  
668  
669  
670  
671  
672  
673  
674  
675  
676  
677  
678  
679  
680  
681  
682  
683  
684  
685  
686  
687  
688  
689  
690  
691  
692  
693  
694  
695  
696  
697  
698  
699  
700  
701  
702  
703  
704  
705  
706  
707  
708  
709  
710  
711  
712  
713  
714  
715  
716  
717  
718  
719  
720  
721  
722  
723  
724  
725  
726  
727  
728  
729  
730  
731  
732  
733  
734  
735  
736  
737  
738  
739  
740  
741  
742  
743  
744  
745  
746  
747  
748  
749  
750  
751  
752  
753  
754  
755  
756  
757  
758  
759  
760  
761  
762  
763  
764  
765  
766  
767  
768  
769  
770  
771  
772  
773  
774  
775  
776  
777  
778  
779  
780  
781  
782  
783  
784  
785  
786  
787  
788  
789  
790  
791  
792  
793  
794  
795  
796  
797  
798  
799  
800  
801  
802  
803  
804  
805  
806  
807  
808  
809  
810  
811  
812  
813  
814  
815  
816  
817  
818  
819  
820  
821  
822  
823  
824  
825  
826  
827  
828  
829  
830  
831  
832  
833  
834  
835  
836  
837  
838  
839  
840  
841  
842  
843  
844  
845  
846  
847  
848  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
860  
861  
862  
863  
864  
865  
866  
867  
868  
869  
870  
871  
872  
873  
874  
875  
876  
877  
878  
879  
880  
881  
882  
883  
884  
885  
886  
887  
888  
889  
890  
891  
892  
893  
894  
895  
896  
897  
898  
899  
900  
901  
902  
903  
904  
905  
906  
907  
908  
909  
910  
911  
912  
913  
914  
915  
916  
917  
918  
919  
920  
921  
922  
923  
924  
925  
926  
927  
928  
929  
930  
931  
932  
933  
934  
935  
936  
937  
938  
939  
940  
941  
942  
943  
944  
945  
946  
947  
948  
949  
950  
951  
952  
953  
954  
955  
956  
957  
958  
959  
960  
961  
962  
963  
964  
965  
966  
967  
968  
969  
970  
971  
972  
973  
974  
975  
976  
977  
978  
979  
980  
981  
982  
983  
984  
985  
986  
987  
988  
989  
990  
991  
992  
993  
994  
995  
996  
997  
998  
999  
1000  
1001  
1002  
1003  
1004  
1005  
1006  
1007  
1008  
1009  
1010  
1011  
1012  
1013  
1014  
1015  
1016  
1017  
1018  
1019  
1020  
1021  
1022  
1023  
1024  
1025  
1026  
1027  
1028  
1029  
1030  
1031  
1032  
1033  
1034  
1035  
1036  
1037  
1038  
1039  
1040  
1041  
1042  
1043  
1044  
1045  
1046  
1047  
1048  
1049  
1050  
1051  
1052  
1053  
1054  
1055  
1056  
1057  
1058  
1059  
1060  
1061  
1062  
1063  
1064  
1065  
1066  
1067  
1068  
1069  
1070  
1071  
1072  
1073  
1074  
1075  
1076  
1077  
1078  
1079  
1080  
1081  
1082  
1083  
1084  
1085  
1086  
1087  
1088  
1089  
1090  
1091  
1092  
1093  
1094  
1095  
1096  
1097  
1098  
1099  
1100  
1101  
1102  
1103  
1104  
1105  
1106  
1107  
1108  
1109  
1110  
1111  
1112  
1113  
1114  
1115  
1116  
1117  
1118  
1119  
1120  
1121  
1122  
1123  
1124  
1125  
1126  
1127  
1128  
1129  
1130  
1131  
1132  
1133  
1134  
1135  
1136  
1137  
1138  
1139  
1140  
1141  
1142  
1143  
1144  
1145  
1146  
1147  
1148  
1149  
1150  
1151  
1152  
1153  
1154  
1155  
1156  
1157  
1158  
1159  
1160  
1161  
1162  
1163  
1164  
1165  
1166  
1167  
1168  
1169  
1170  
1171  
1172  
1173  
1174  
1175  
1176  
1177  
1178  
1179  
1180  
1181  
1182  
1183  
1184  
1185  
1186  
1187  
1188  
1189  
1190  
1191  
1192  
1193  
1194  
1195  
1196  
1197  
1198  
1199  
1200  
1201  
1202  
1203  
1204  
1205  
1206  
1207  
1208  
1209  
1210  
1211  
1212  
1213  
1214  
1215  
1216  
1217  
1218  
1219  
1220  
1221  
1222  
1223  
1224  
1225  
1226  
1227  
1228  
1229  
1230  
1231  
1232  
1233  
1234  
1235  
1236  
1237  
1238  
1239  
1240  
1241  
1242  
1243  
1244  
1245  
1246  
1247  
1248  
1249  
1250  
1251  
1252  
1253  
1254  
1255  
1256  
1257  
1258  
1259  
1260  
1261  
1262  
1263  
1264  
1265  
1266  
1267  
1268  
1269  
1270  
1271  
1272  
1273  
1274  
1275  
1276  
1277  
1278  
1279  
1280  
1281  
1282  
1283  
1284  
1285  
1286  
1287  
1288  
1289  
1290  
1291  
1292  
1293  
1294  
1295  
1296  
1297  
1298  
1299  
1300  
1301  
1302  
1303  
1304  
1305  
1306  
1307  
1308  
1309  
1310  
1311  
1312  
1313  
1314  
1315  
1316  
1317  
1318  
1319  
1320  
1321  
1322  
1323  
1324  
1325  
1326  
1327  
1328  
1329  
1330  
1331  
1332  
1333  
1334  
1335  
1336  
1337  
1338  
1339  
1340  
1341  
1342  
1343  
1344  
1345  
1346  
1347  
1348  
1349  
1350  
1351  
1352  
1353  
1354  
1355  
1356  
1357  
1358  
1359  
1360  
1361  
1362  
1363  
1364  
1365  
1366  
1367  
1368  
1369  
1370  
1371  
1372  
1373  
1374  
1375  
1376  
1377  
1378  
1379  
1380  
1381  
1382  
1383  
1384  
1385  
1386  
1387  
1388  
1389  
1390  
1391  
1392  
1393  
1394  
1395  
1396  
1397  
1398  
1399  
1400  
1401  
1402  
1403  
1404  
1405  
1406  
1407  
1408  
1409  
1410  
1411  
1412  
1413  
1414  
1415  
1416  
1417  
1418  
1419  
1420  
1421  
1422  
1423  
1424  
1425  
1426  
1427  
1428  
1429  
1430  
1431  
1432  
1433  
1434  
1435  
1436  
1437  
1438  
1439  
1440  
1441  
1442  
1443  
1444  
1445  
1446  
1447  
1448  
1449  
1450  
1451  
1452  
1453  
1454  
1455  
1456  
1457  
1458  
1459  
1460  
1461  
1462  
1463  
1464  
1465  
1466  
1467  
1468  
1469  
1470  
1471  
1472  
1473  
1474  
1475  
1476  
1477  
1478  
1479  
1480  
1481  
1482  
1483  
1484  
1485  
1486  
1487  
1488  
1489  
1490  
1491  
1492  
1493  
1494  
1495  
1496  
1497  
1498  
1499  
1500  
1501  
1502  
1503  
1504  
1505  
1506  
1507  
1508  
1509  
1510  
1511  
1512  
1513  
1514  
1515  
1516  
1517  
1518  
1519  
1520  
1521  
1522  
1523  
1524  
1525  
1526  
1527  
1528  
1529  
1530  
1531  
1532  
1533  
1534  
1535  
1536  
1537  
1538  
1539  
1540  
1541  
1542  
1543  
1544  
1545  
1546  
1547  
1548  
1549  
1550  
1551  
1552  
1553  
1554  
1555  
1556  
1557  
1558  
1559  
1560  
1561  
1562  
1563  
1564  
1565  
1566  
1567  
1568  
1569  
1570  
1571  
1572  
1573  
1574  
1575  
1576  
1577  
1578  
1579  
1580  
1581  
1582  
1583  
1584  
1585  
1586  
1587  
1588  
1589  
1590  
1591  
1592  
1593  
1594  
1595  
1596  
1597  
1598  
1599  
1600  
1601  
1602  
1603  
1604  
1605  
1606  
1607  
1608  
1609  
1610  
1611  
1612  
1613  
1614  
1615  
1616  
1617  
1618  
1619  
1620  
1621  
1622  
1623  
1624  
1625  
1626  
1627  
1628  
1629  
1630  
1631  
1632  
1633  
1634  
1635  
1636  
1637  
1638  
1639  
1640  
1641  
1642  
1643  
1644  
1645  
1646  
1647  
1648  
1649  
1650  
1651  
1652  
1653  
1654  
1655  
1656  
1657  
1658  
1659  
1660  
1661  
1662  
1663  
1664  
1665  
1666  
1667  
1668  
1669  
1670  
1671  
1672  
1673  
1674  
1675  
1676  
1677  
1678  
1679  
1680  
1681  
1682  
1683  
1684  
1685  
1686  
1687  
1688  
1689  
1690  
1691  
1692  
1693  
1694  
1695  
1696  
1697  
1698  
1699  
1700  
1701  
1702  
1703  
1704  
1705  
1706  
1707  
1708  
1709  
1710  
1711  
1712  
1713  
1714  
1715  
1716  
1717  
1718  
1719  
1720  
1721  
1722  
1723  
1724  
1725  
1726  
1727  
1728  
1729  
1730  
1731  
1732  
1733  
1734  
1735  
1736  
1737  
1738  
1739  
1740  
1741  
1742  
1743  
1744  
1745  
1746  
1747  
1748  
1749  
1750  
1751  
1752  
1753  
1754  
1755  
1756  
1757  
1758  
1759  
1760  
1761  
1762  
1763  
1764  
1765  
1766  
1767  
1768  
1769  
1770  
1771  
1772  
1773  
1774  
1775  
1776  
1777  
1778  
1779  
1780  
1781  
1782  
1783  
1784  
1785  
1786  
1787  
1788  
1789  
1790  
1791  
1792  
1793  
1794  
1795  
1796  
1797  
1798  
1799  
1800  
1801  
1802  
1803  
1804  
1805  
1806  
1807  
1808  
1809  
1810  
1811  
1812  
1813  
1814  
1815  
1816  
1817  
1818  
1819  
1820  
1821  
1822  
1823  
1824  
1825  
1826  
1827  
1828  
1829  
1830  
1831  
1832  
1833  
1834  
1835  
1836  
1837  
1838  
1839  
1840  
1841  
1842  
1843  
1844  
1845  
1846  
1847  
1848  
1849  
1850  
1851  
1852  
1853  
1854  
1855  
1856  
1857  
1858  
1859  
1860  
1861  
1862  
1863  
1864  
1865  
1866  
1867  
1868  
1869  
1870  
1871  
1872  
1873  
1874  
1875  
1876  
1877  
1878  
1879  
1880  
1881  
1882  
1883  
1884  
1885  
1886  
1887  
1888  
1889  
1890  
1891  
1892  
1893  
1894  
1895  
1896  
1897  
1898  
1899  
1900  
1901  
1902  
1903  
1904  
1905  
1906  
1907  
1908  
1909  
1910  
1911  
1912  
1913  
1914  
1915  
1916  
1917  
1918  
1919  
1920  
1921  
1922  
1923  
1924  
1925  
1926  
1927  
1928  
1929  
1930  
1931  
1932  
1933  
1934  
1935  
1936  
1937  
1938  
1939  
1940  
1941  
1942  
1943  
1944  
1945  
1946  
1947  
1948  
1949  
1950  
1951  
1952  
1953  
1954  
1955  
1956  
1957  
1958  
1959  
1960  
1961  
1962  
1963  
1964  
1965  
1966  
1967  
1968  
1969  
1970  
1971  
1972  
1973  
1974  
1975  
1976  
1977  
1978  
1979  
1980  
1981  
1982  
1983  
1984  
1985  
1986  
1987  
1988  
1989  
1990  
1991  
1992  
1993  
1994  
1995  
1996  
1997  
1998  
1999  
2000  
2001  
2002  
2003  
2004  
2005  
2006  
2007  
2008  
2009  
2010  
2011  
2012  
2013  
2014  
2015  
2016  
2017  
2018  
2019  
2020  
2021  
2022  
2023  
2024  
2025  
2026  
2027  
2028  
2029  
2030  
2031  
2032  
2033  
2034  
2035  
2036  
2037  
2038  
2039  
2040  
2041  
2042  
2043  
2044  
2045  
2046  
2047  
2048  
2049  
2050  
2051  
2052  
2053  
2054  
2055  
2056  
2057  
2058  
2059  
2060  
2061  
2062  
2063  
2064  
2065  
2066  
2067  
2068  
2069  
2070  
2071  
2072  
2073  
2074  
2075  
2076  
2077  
2078  
2079  
2080  
2081  
2082  
2083  
2084  
2085  
2086  
2087  
2088  
2089  
2090  
2091  
2092  
2093  
2094  
2095  
2096  
2097  
2098  
2099  
2100  
2101  
2102  
2103  
2104  
2105  
2106  
2107  
2108  
2109  
2110  
2111  
2112  
2113  
2114  
2115  
2116  
2117  
2118  
2119  
2120  
2121  
2122  
2123  
2124  
2125  
2126  
2127  
2128  
2129  
2130  
2131  
2132  
2133  
2134  
2135  
2136  
2137  
2138  
2139  
2140  
2141  
2142  
2143  
2144  
2145  
2146  
2147  
2148  
2149  
2150  
2151  
2152  
2153  
2154  
2155  
2156  
2157  
2158  
2159  
2160  
2161  
2162  
2163  
2164  
2165  
2166  
2167  
2168  
2169  
2170  
2171  
2172  
2173  
2174  
2175  
2176  
2177  
2178  
2179  
2180  
2181  
2182  
2183  
2184  
2185  
2186  
2187  
2188  
2189  
2190  
2191  
2192  
2193  
2194  
2195  
2196  
2197  
2198  
2199  
2200  
2201  
2202  
2203  
2204  
2205  
2206  
2207  
2208  
2209  
2210  
2211  
221

## BRIEF DESCRIPTION OF THE DRAWINGS

The foregoing aspects and many of the attendant advantages of this invention will become more readily appreciated as the same becomes better understood by reference to the following detailed description, when taken in conjunction with the accompanying drawings, wherein:

FIGURE 1 is a block diagram illustrating the disambiguation token of the present invention;

FIGURES 2A and 2B illustrate various types of memory locations (LOCs) corresponding to memory references and function calls for an exemplary function;

FIGURE 2C illustrates a local and global LOC set corresponding to memory references in an exemplary function;

FIGURE 3 is a block diagram illustrating various components of a disambiguation token corresponding to a direct memory reference;

FIGURE 4 is a block diagram illustrating various components of a disambiguation token corresponding to an indirect memory reference;

FIGURE 5 is a block diagram of an exemplary compiler that implements the present invention;

FIGURE 6 is a block diagram illustrating the various modules used during memory disambiguation and the interfaces between them.

FIGURE 7A-D collectively comprise a flowchart illustrating the logic used by the present invention during a compilation process that implements the disambiguation method of the present invention;

FIGURES 8A-C collectively comprise a flowchart illustrating the logic used by the disambiguator module of the present invention during the compilation process; and

FIGURE 9 is a block diagram of an exemplary computer system on which the present invention can be implemented.

## DETAILED DESCRIPTION OF THE ILLUSTRATED EMBODIMENTS

A method and system for memory disambiguation is described in detail herein. In the following description, numerous specific details are provided to provide a thorough understanding of embodiments of the invention. One skilled in the relevant art will recognize, however, that the invention can be practiced without one or more of the specific details, or with other methods, components, etc. In other instances, well-known structures or operations are not shown or described in detail to avoid obscuring aspects of various embodiments of the invention.

Reference throughout this specification to “one embodiment” or “an embodiment” means that a particular feature, structure, or characteristic described in connection with the embodiment is included in at least one embodiment of the present invention. Thus, the appearances of the phrases “in one embodiment” or “in an embodiment” in various places throughout this specification are not necessarily all referring to the same embodiment. Furthermore, the particular features, structures, or characteristics may be combined in any suitable manner in one or more embodiments.

One aspect of the present invention comprises a novel memory disambiguation method that provides accurate memory disambiguation that is efficient in compile time and memory usage. The method preserves high-level semantics and other information necessary for disambiguation in a new structure called a ***disam token***. The disam token and a symbolic memory reference representation associated with it are also the means by which the various memory disambiguation modules and their clients communicate, forming the basis of a complete memory disambiguation system. An algorithm for creating and maintaining the disam tokens and disambiguation information and an algorithm for applying various disambiguation rules that utilize the information are discussed in detail below.

Disam tokens are created for each memory reference after the interprocedural analysis and optimization (including inlining) is done, but before the optimization of each

individual function. Alternatively, disam tokens could be created before interprocedural analysis and optimization. A unique disam token is associated with every memory reference in the function that the compiler is currently processing. The disam token provides access to all the information (either directly or through other links) necessary to perform memory disambiguation. Examples of disam tokens include, but are not limited to, a data structure embedded in the memory reference operators of the intermediate language (IL) or a separate data structure linked to the memory reference operator via a pointer or hash table lookup.

The relationship between a memory reference, its disam token, a symbolic memory reference representation, and other information needed for memory disambiguation are illustrated in FIGURE 1. Each memory reference is associated with a disam token. Each disam token includes a plurality of links (e.g., pointers) to where various information pertaining to the disam token and its use are stored, including a LOC set, parameter information, type information, a data dependence key, a flow-sensitive points-to, and base + offset information. As described in further detail below, LOC set contains a pointer to a symbol table entry.

Memory references are represented symbolically using a data structure called a **LOC set**. There are several types of LOCs corresponding to the various types of storage locations, as illustrated in FIGURES 2A-B. For example there are LOCs representing global variables, local variables, formal and actual function parameters, registers, dynamically allocated heap objects, and even the text of a function.

The contents of the LOC set vary depending on the type of memory reference. As shown in FIGURE 3, for direct memory references, LOC set contains a single LOC representing the memory object (local or global variable) that is accessed. For indirect memory reference, LOC set contains a single LOC representing a

pointer and the dereference level, as depicted in FIGURE 4. The LOC provides access to the symbol table information associated with the memory object accessed (direct references) or the pointer (indirect references). FIGURE 2C shows LOC sets for memory references with various levels of dereference. The dereference level is represented by a dereference mask. Bit position 0 in the mask represents the address of operator (&), position 1 represents a direct memory reference, position 2 represents an indirect reference with dereference level 1 (1 star), position 2 dereference level 2 (2 stars) and so on. Figure 2C shows the dereference masks in binary.

The disam token also contains a link to the type information 20 for the memory reference. For array references, the disam token contains a data dependence key 22 that is used to access a table of array data dependence information 46. For Indirect references, the disam token provides an interface to flow-sensitive points-to information 24. This information must be stored for each memory reference rather than for each pointer. Finally, the disam token also contains information about parameters and copies of parameters, as represented by parameter information 18, and base and offset information for low-level disambiguation, as represented by base + offset information 26.

Disam tokens are created early in the compiler while the memory references are still in a form that is similar to the source code and before variables are promoted to registers so as not to lose the symbol table information for pointers that get registerized. All loads and stores eventually become indirect off registers and it is hard to determine at that point whether the memory reference was originally direct or indirect.

Forward substitution can have an effect similar to copy and constant propagation. For example, the following sequence of code:

```
t = &a
foo (t[i]);
```

would become:



foo(a[i]);

after forward substitution. `t[i]` is an indirect reference while `a[i]` is a direct reference.

- 5 For reasons of both performance and correctness, the disam token information must be updated to reflect this transformation.

Disam tokens must be maintained whenever memory references are created, copied, or translated to a different form as shown in the process above. Whenever a memory reference is copied by an optimization, the associated disam token is automatically copied. At selected points during the compilation, the disam tokens are  
10 verified to make sure that there is still a token for each memory reference and that the contents of the token look reasonable. The purpose of this is to catch any errors in the maintenance of the disam tokens.

With reference to FIGURE 5, an exemplary compiler architecture 50 in which the present invention may be implemented includes a front-end 52, an optimizer 54, and a code generator 56, as well as other conventional compiler blocks that are not shown. In addition to performing conventional compilation optimizations, optimizer 54 performs the memory disambiguation method of the invention using a disambiguation server 58 that provides disambiguation services to various optimization clients, including high  
15 level optimizer (HLO) clients 82, scalar optimizer clients 80, and code generator clients 60.

FIGURE 6 shows a block diagram of the various modules involved in memory disambiguation and the interfaces between them. A disambiguator module 62 receives queries from a client, queries the other modules if necessary, interprets all the  
20 information, and returns a disambiguation result. Note that both the sources of disambiguation information and the clients operate at various levels of abstraction in the compiler. For example points-to and MOD/REF analysis occur early during

interprocedural analysis, array data dependence analysis occurs in the middle of the compilation after loops have been unrolled, and base and offset analysis occurs late after the memory references have been translated to their lowest level form. The clients range from the high level optimizer to the code schedulers. What allows the disambiguator to communicate with them all is the disam token and LOC framework. With the exception of the base and offset analysis, the disambiguator views the memory references in the same way throughout the compilation. Simply by looking at the disam token, the low level loads and stores that the scheduler wants to reorder are translated to a form that the high-level points-to analysis and symbol table understand. Clients pass the two memory references to the disambiguator using disam tokens which are independent of the different ILs used by the optimizer and code generator.

As shown in FIGURE 6, disambiguator 62 interacts with a plurality of modules that are internal to disambiguation server 58, including an array data dependence table 64, a flow-insensitive points-to module 66, a base + offset analysis module 68, a flow-sensitive points-to module 70, a parameter copy and modification analysis module 72, and a function call mod/ref module 74. Disambiguator 58 also interacts with several external (to disambiguation server 58) modules, including a symbol table 76, various schedulers 78, various optimizer clients 80, and high-level optimizer (HLO) clients 82.

If both memory references are direct (note that direct vs. indirect is easily determined from the LOC set representation of the memory reference), their LOC sets are compared to determine whether or not the same memory object is accessed. LOCs are created in such a way that if the LOCs are different, then different memory objects are accessed. If the same object is accessed, the disambiguator then attempts to determine if overlapping portions of the object are accessed. From the symbol table information, the disambiguator can find out the type of the high-level object accessed, such as a scalar, array, or record (structure). For example, the array data dependence

information is used to determine if the same array element is accessed. For example, for array references, the disam token contains a key that is used to access a table of data dependence information. For two references to the same array object, a table lookup is done using the two keys. The result of the lookup is an indication of whether or not there is a dependence between the two array references and the characteristics of that dependence. In addition to array references, the data dependence key and table can be used to encode information about dependences between any two memory references. For example, in loops containing directives to ignore dependences, the data dependence keys and table are used to encode information for any pairs of memory references that the disambiguator is not able to disambiguate without using the directive. Structure type information from the symbol table is used to determine if overlapping fields of a structure are accessed. This information is generated by the front-end and is attached to the memory references in the IL. This information is stored in the disam token when the memory references are translated to the code generator's IL. The type information contains the type and offset information for the field within the structure.

Compiler generated references can often be easily disambiguated from all other memory references. For example, references to read only storage areas can be disambiguated from all stores. The Itanium™ software conventions require several forms of read only objects, notably for function pointers and for global variable accessing. The disambiguator can trivially prove these references independent.

If at least one of the memory references is indirect, the disambiguator first attempts to prove independence without knowing where the indirect references point to. The LOC for the pointer is used to look up the symbol table information for that pointer. The disambiguator also maintains a table of information about parameters and copies of parameters. This information is stored in a hashtable indexed by the LOC. For example, an indirect reference off an unmodified parameter or a copy of that parameter

could not possibly access a stack allocated local variable from the function in which the two references appear. When the compiler is run with interprocedural optimization, it has the ability to automatically detect that it is seeing the whole program. That is it can detect whether or not there are calls to functions that it has not seen and does not know the behavior of. When the compiler can see the whole program, the disambiguator knows that an indirect reference cannot possibly access a global variable that has not had its address taken. Address taken information is available through the symbol table.

Next, the disambiguator turns to a method that utilizes the lowered addressing. It analyzes the address expression of each memory reference and tries determine a base and offset. If successful it caches the information in the disam token and compares the base and offset for the two memory references. If they have the same base, the disambiguator can use the offsets and sizes of the memory references to determine whether or not they overlap.

If simple rules such as those above do not allow the disambiguator to prove independence of the memory references, the results of points-to analysis are consulted. For each memory reference, the disambiguator passes the LOC set representing the memory reference to the points-to interface, which returns a LOC set representing the set of locations that could be accessed by that memory reference. The disambiguator then compares the LOC sets to determine if there is any overlap. In the case of flow-sensitive points-to, the disam token contains the points-to LOC set.

Flow-insensitive points-to analysis is conducted based on summary information collected before procedure inlining. However, as the inliner makes a copy of the callee function to insert in the caller, it converts the local variables in the callee into new local variables in the caller. Disam tokens are created after inlining and therefore the LOCs are created for the new local variables in the caller rather the variables in the original copy of the callee. Because the new local variable in the caller (and the corresponding LOC) did not exist at the time that points-to analysis was done and the key to obtain the

points-to set of a variable is the LOC representing the pointer, we are not able to obtain the points-to sets of the new local variables in the caller. To solve this problem, we keep a pointer in each variable data structure, to the "original" LOC corresponding to it. While converting a local variable of the callee into a local variable of the caller during inlining, we initialize the original\_LOC pointer of the new variable to the original\_LOC pointer of the original variable. This enables the disambiguator to obtain the original LOC representing the local variable in the original copy of the caller and query the points-to interface. When not querying points to information, the disambiguator uses the LOC representing the new local variable. Thus when there are two copies of the same callee inlined at two different call sites within the same caller, there are two different sets of new local variables and the disambiguator can distinguish between them.

Finally, the disambiguator can perform type-based disambiguation based on the languages type aliasability rules. For example, under the ANSI C type aliasability rules, a reference to an object of type float cannot overlap with a reference to an object of type integer.

As discussed above, disamb tokens are also associated with all function calls. Clients can query the disambiguator with the tokens for a memory reference and a function call. The disambiguator passes a LOC set representing the function call (recall that LOCs can represent functions) to the MOD/REF module and receives a LOC set representing the set of memory locations that could be modified (written) or referenced (read) as a result of the function call. In the MOD/REF module, the compiler performs some kind of mod/ref analysis, which comprises determining the set of memory location modified (written) or referenced (read) by each function. This could be as simple as knowing that certain library functions do not modify or reference any user program variables, or as complex as a full interprocedural analysis. The set of locations modified or referenced is represented as a mod or ref LOC set respectively. These are stored for in the MOD/REF module for later use by the disambiguator. For indirect

calls, there is a LOC set representing the dereferenced pointer. The points-to interface is then queried to determine the set of functions that could be called. The MOD or REF sets for these functions are unioned across the different functions in the set. The disambiguator then intersects the LOC set for the memory reference with the MOD or REF set for the function call to determine if the function call reads or writes any of the same memory locations accessed by the memory reference.

Another capability of this disambiguation method is the ability to compute the address relationship between a pair of memory references. This information is needed for the compiler to optimize around memory system limitations such as cache bank or store buffer conflicts. The information can be used by the schedulers to compute artificial dependences for scheduling around memory system limitations and to do post-increment optimization. Also, it can be used by the high-level optimizer to coalesce loads and stores (combine a sequence of small loads or stores into fewer larger loads or stores). Computation of address relations is similar to determination of overlap except that instead of returning dependent or independent, the disambiguator uses the information in the disambiguator tokens to compute the difference in starting addresses of two memory references and the alignment of the two memory references.

Another capability of this disambiguation method is to determine the exact nature of the overlap between memory references. For example, using the information in the disambiguation token, it can determine if one memory reference overlaps exactly with another (same starting address and same size) or if one memory reference is a subset of the other. This information can be used by the optimizer to generate the code needed to perform store forwarding in the case of a store followed by a load of a subset of the bytes stored.

In general, a compiler that implements the present invention will perform a conventional compilation process augmented with various functions corresponding to the memory disambiguation method of the invention. With reference to the flowchart

illustrated in FIGURES 7A-D, the logic implemented by such a compiler for collection, maintenance, and use of disambiguation information during a compilation process is illustrated, wherein conventional compilation functions are depicted as boxes with non-  
5 bolded text, while functions pertaining to the memory disambiguation functions provided by the invention are depicted in boxes with bolded text.

As indicated by start and end loop blocks 100 and 102, the compilation process begins by performing some initialization functions on each source file that is part of the compilation, including a front-end analysis, as provided by a block 104. The front-end analysis includes lexical and syntactic analysis, creation of the symbol table, semantic  
10 analysis, and other common front-end functions that are well-known in the art. As indicated by start and loop block 106 and 108, for each function in the current file an original LOC is created for the left-hand side and right-hand side of each assignment in a block 110, and points-to basis assignments are created in a block 112.

After the initialization functions have been applied to the source files, the symbol tables and point-to basis from the files are combined in a block 114. A points-to  
15 analysis is then performed in a block 116. This comprises the processing of the points-to basis assignments for each function or across all functions and building a points-to graph that describes the set of memory objects accessible through each pointer. As identified by a start loop block 118 and an end loop block 120 in FIGURE 7D, a set of  
20 functions described in the following paragraphs are then applied to each function.

In a block 122, conventional procedure integration is performed. This will typically comprise inlining and partial inlining of procedures and functions. Next, a disamb token for each memory reference is created in a block 124, while new LOCs for local memory references from the inlined routines are created in a block 126.

25 With reference to FIGURE 7B, the flowchart continues in a block 128 in which a forward substitution and indirect to direct reference conversion is performed. This is particularly important for Fortran and C++ by-reference parameters that can become

direct references after inlining. As provided by start and end loop blocks 130 and 132, for each indirect reference that is made into a direct reference by substitution, a corresponding disam token is updated to represent a direct reference instead of the previous indirect reference, as provided by a block 134. Next, in a decision block 136 a determination is made to whether the function has any local scalar variables whose address is not referred to. If the answer is yes, the logic proceeds to a block 138 in which such local scalar variables are promoted to registers for the entire life of the function.

A first set of conventional optimization phases are performed in a block 140. The optimization phases shown in the Figures are intended to be examples. As will be recognized by those skilled in the art, the number and type of optimizations phases may vary, depending on the particular implementation. Next, in a block 142, the high-level optimizer 82 queries disambiguator 62 when building dependence graphs. Dead code elimination is then performed in a block 144, which includes using the disam tokens to determine the set of local memory objects that are not referenced after they have been modified, as provided by a block 146. A second set of conventional optimization phases are then performed in a block 148, and the flowchart advances to a block 150 in FIGURE 7C.

In block 150, loads of large constants are materialized. This includes creating disam tokens for new loads, as provided by a block 152. Loads and stores for parameter passing are then materialized in a block 154, which includes creating disam tokens for new memory references in a block 156. Memory references are then translated to a lower-level form in a block 158, which includes copying disam tokens from old to new memory references in a block 160. A third set of optimization phases are then performed in a block 162.

The logic next proceeds to a block 164 in which the disam token for each memory reference is verified. A fourth set of optimization phases are then performed in



a block 166. Partial redundancy elimination (PRE) is next performed in a block 168, which includes querying disambiguator 62 to determine if stores kill (i.e., overlap with) available loads.

With reference to FIGURE 7D, the logic next proceeds to a block 172 in which partial dead store elimination is performed. As provided by a block 174, this includes querying disambiguator 62 if stores or loads kill any later stores. A fifth set of optimization phases are then performed in a block 176.

Next, the disam token for each memory reference is verified in a block 178. The program is then translated from the optimizer to code generator IL in a block 180, which includes maintaining a pointer from each load or store to a corresponding disam token, as provided by a block 182. A sixth set of optimization phases is then performed in a block 184.

The compiler then performs code scheduling, which includes querying disambiguator 62 to determine if two memory references access overlapping memory locations, as provided by blocks 186 and 188. Processing of the current function is completed by performing register allocation and assembly or object code emission in a block 189. The logic then loops back to block 118 to begin processing the next function. Processing of each function in a similar manner to that described above is continued until all of the functions have been processed, thereby completing the compilation process, as indicated by a block 190.

Details of the memory disambiguation process are shown in the flowchart of FIGURES 8A-C. With respect to the flowchart and the following discussion, a disambiguation process as applied to two memory references is presented. With reference to a decision block 200, a determination is made to whether both memory references are direct. This can be easily determined from the LOC set representation of the memory reference. If both memory references are direct, the logic proceeds to a decision block 202, in which the LOC sets are compared to determine whether or not

the same memory object is accessed. LOCs are created in such a way that if the LOCs are different, then different memory objects are accessed. If the two LOCs are different, the disambiguator determines that the memory references are independent, as indicated in a return block 204.

5 If the same object is accessed, as indicated by a no answer to decision block 202, the disambiguator then attempts to determine if overlapping portions of the object are accessed. Accordingly, the logic proceeds to a decision block 206 comprising a switch statement that redirects the process flow based on whether the memory object is a scalar, a record (i.e., data structure), or an array, as depicted by  
10 switch case blocks 208, 210, and 212, respectively. From the symbol table information, the disambiguator can determine the type of high-level object being accessed.

If the memory object is a scalar, data indicating that the memory references are dependent is returned in a block 214. If the memory object is a record, type information for the memory object is retrieved, a check is made to see if an overlap within the record exists, and the results of the type information and overlap check results are  
15 returned in a block 216. Structure type information from the symbol table is used to determine if overlapping fields of a structure are accessed. This information is generated by the front-end analysis provided in block 104 above, and is attached to the memory references in the IL. This information is stored in the disam token when the  
20 memory references are translated to the code generator's IL. The type information contains the type and offset information for the field within the structure.

If the memory object is an array, the array data dependence information is used to determine if the same array element is accessed. For array references, the disam token contains a key (data dependence key 22) that is used to access a table of array  
25 data dependence information. For two references to the same array object, a table lookup is done using the two keys. The result of the lookup is an indication of whether

or not there is a dependence between the two array references and the -characteristics of that dependence. The result of this determination is returned in a block 218.

If at least one of the memory references is indirect (as indicated by a no answer to decision block 200), the logic proceeds to a decision block 220, in which a determination is made to whether both references are indirect. If only one of the two references is indirect, the logic proceeds to blocks 222 and 224, in which properties for the direct reference, and properties for the pointer for the indirect reference are obtained from the symbol table. In the latter case, the LOC for the pointer is used to look up the symbol table information for that pointer. With reference to FIGURE 8B, a determination is then made in a decision block 226 as to whether the pointer could possibly point to a directly accessed variable. As discussed above, an indirect reference off an unmodified parameter or a copy of that parameter could not possibly access a stack allocated local variable from the function in which the two references appear. If the pointer could not possibly point to the directly accessed variable, then the memory references are determined to be independent, as provided by a return block 228.

Returning to decision block 220, if both memory references are indirect (i.e., they both are pointers), the logic proceeds to a block 230 in which properties for both of the pointers are obtained from the symbol table. In a decision block 232 a determination is then made to whether the properties indicate that the two pointers could possibly access overlapping memory locations. If this determination is false, the disambiguator returns a result in a return block 234 indicating the memory references are independent.

If the determination for either of decision blocks 226 or 232 is yes, the logic proceeds to a block 235 in which a data dependence table lookup is done if the two memory references each have a valid data dependence key. The data dependence table lookup returns either independent, dependent, or don't know. As indicated by a

decision block 236, if the result is known, the data dependence table lookup result is returned in a return block 238.

If the table lookup returns don't know, base and offset information is obtained in a block 240, and a determination is made in a decision block 242 to whether or not both memory references share the same base address. If they do, their offsets and sizes are compared to see if an overlap exists, and the results are returned in a block 244. If they do not share the same base address, the logic proceeds to a decision block 246 in which a determination is made to whether the points-to analysis has already been run. If it has, the points-to LOC sets for both memory references are obtained in a block 248, and the LOC sets are compared in a block 250. In a decision block 252, a determination is made to whether an intersection exists between the LOC sets. If no intersection exists, the memory references are independent, as indicated by a return block 254.

If either the point-to analysis has not been performed, or an intersection is found in decision block 252, the logic proceeds to obtain type and parameter information for both memory references, as provided by a block 256. Language type aliasability rules and other language rules are then applied, with the results being returned in a block 258. As described above, aliasability rules are used to determine whether certain object types can overlap with one another. If they can, the memory references are dependent. If they cannot, the memory references are independent. In the Fortran language, distinct by-reference parameters are always independent.

### **Exemplary Computer System for implementing the Invention**

With reference to FIGURE 9, a generally conventional computer 300 is illustrated, which is suitable for use in connection with practicing the present invention, and may be used for running a client application comprising one or more software modules that implement the various functions of the invention discussed above.

Examples of computers that may be suitable for clients as discussed above include PC-class systems operating the Windows NT or Windows 2000 operating systems, Sun workstations operating the UNIX-based Solaris operating system, and various computer architectures that implement LINUX operating systems. Alternatively, other similar types of computers may be used, including computers with multiple processors. The computer may also be a server, such as a Hewlett Packard Netserver, an IBM Netfinity server, various servers made by Dell and Compaq, as well as UNIX-based servers and LINUX-based servers.

Computer 300 includes a processor chassis 302 in which are mounted a floppy disk drive 304, a hard drive 306, a motherboard populated with appropriate integrated circuits (not shown) including memory and one or more processors, and a power supply (also not shown), as are generally well known to those of ordinary skill in the art. It will be understood that hard drive 306 may comprise a single unit, or multiple hard drives, and may optionally reside outside of computer 300. A monitor 308 is included for displaying graphics and text generated by software programs and program modules that are run by the computer. A mouse 310 (or other pointing device) may be connected to a serial port (or to a bus port or USB port) on the rear of processor chassis 302, and signals from mouse 310 are conveyed to the motherboard to control a cursor on the display and to select text, menu options, and graphic components displayed on monitor 308 by software programs and modules executing on the computer. In addition, a keyboard 312 is coupled to the motherboard for user entry of text and commands that affect the running of software programs executing on the computer. Computer 300 may also include a network interface card (not shown) for connecting the computer to a computer network, such as a local area network, wide area network, or the Internet.

Computer 300 may also optionally include a compact disk-read only memory (CD-ROM) drive 314 into which a CD-ROM disk may be inserted so that executable

files and data on the disk can be read for transfer into the memory and/or into storage on hard drive 306 of computer 300. Other mass memory storage devices such as an optical recorded medium or DVD drive may be included. The machine instructions comprising the software program that causes the CPU to implement the functions of the present invention that have been discussed above will likely be distributed on floppy disks or CD-ROMs (or other memory media) and stored in the hard drive until loaded into random access memory (RAM) for execution by the CPU. Optionally, the machine instructions may be loaded via a computer network.

Although the present invention has been described in connection with a preferred form of practicing it and modifications thereto, those of ordinary skill in the art will understand that many other modifications can be made to the invention within the scope of the claims that follow. Accordingly, it is not intended that the scope of the invention in any way be limited by the above description, but instead be determined entirely by reference to the claims that follow.